

.NET Best Practices



Trinay Technology Solution(TTS)

Table of Contents

Improving ASP.NET Performance	5
Objectives	5
Performance and Scalability Issues	5
Consider Security and Performance	6
Partition Your Application Logically	7
Evaluate Affinity	8
Reduce Round Trips	8
Avoid Blocking on Long-Running Tasks	9
Use Caching	9
Avoid Unnecessary Exceptions	10
Resource Management	11
Pool Resources	11
Explicitly Call Dispose or Close on Resources You Open	12
Trim Your Page Size	13
Enable Buffering	14
Use Page.IsPostBack to Minimize Redundant Processing	14
Partition Page Content to Improve Caching Efficiency and Reduce Rendering	15
Ensure Pages Are Batch Compiled	15
Ensure Debug Is Set to False	15
Optimize Expensive Loops	16
Consider Using Server.Transfer Instead of Response.Redirect	16
Use Client-Side Validation	16
Server Controls	16
Identify the Use of View State in Your Server Controls	17
Use Server Controls Where Appropriate	17
Avoid Creating Deep Hierarchies of Controls	18
Data Binding	18
Avoid Using Page.DataBind	18
Minimize Calls to DataBinder.Eval	19

Caching Explained	20
Cache API	20
Output Caching	21
Partial Page or Fragment Caching.....	22
Caching Guidelines.....	22
Separate Dynamic Data from Static Data in Your Pages.....	23
Configure the Memory Limit.....	24
Cache the Right Data.....	24
Refresh Your Cache Appropriately.....	25
Cache the Appropriate Form of the Data	25
Use Output Caching to Cache Relatively Static Pages	25
Choose the Right Cache Location	25
Use VaryBy Attributes for Selective Caching	26
State Management.....	27
Store Simple State on the Client Where Possible	27
Application State.....	28
Use Static Properties Instead of the Application Object to Store Application State	28
Use Application State to Share Static, Read-Only Data	28
Do Not Store STA COM Objects in Application State.....	29
Session State	29
Choosing a State Store	29
Prefer Basic Types to Reduce Serialization Costs.....	30
Disable Session State If You Do Not Use It.....	31
Avoid Storing STA COM Objects in Session State	31
View State.....	31
Disable View State If You Do Not Need It	32
Minimize the Number of Objects You Store In View State.....	32
Determine the Size of Your View State.....	33
tring Management	33
Use Response.Write for Formatting Output.....	33
Use StringBuilder for Temporary Buffers.....	34
Exception Management	34

Implement a Global.asax Error Handler.....	34
Monitor Application Exceptions.....	35
Use Try/Finally on Disposable Resources	35
Write Code That Avoids Exceptions	35
Set Timeouts Aggressively	37
Data Access.....	37
Use Paging for Large Result Sets.....	38
Use a DataReader for Fast and Efficient Data Binding.....	38
Prevent Users from Requesting Too Much Data	39
Consider Caching Data	39
Security Considerations	39
Constrain Unwanted Web Server Traffic.....	39
Turn Off Authentication for Anonymous Access	40
Validate User Input on the Client.....	40
Avoid Per-Request Impersonation.....	40
Avoid Caching Sensitive Data.....	40
Segregate Secure and Non-Secure Content.....	40
Avoid XCOPY Under Heavy Load	41

Improving ASP.NET Performance

Objectives

- Improve page response times.
- Design scalable Web applications.
- Use server controls efficiently.
- Use efficient caching strategies.
- Analyze and apply appropriate state management techniques.
- Minimize view state impact.
- Improve performance without impacting security.
- Minimize COM interop scalability issues.
- Optimize threading.
- Optimize resource management.
- Avoid common data binding mistakes.
- Use security settings to reduce server load.
- Avoid common deployment mistakes.

Performance and Scalability Issues

The main issues that can adversely affect the performance and scalability of your ASP.NET application are summarized below. Subsequent sections in this chapter provide strategies and technical information to prevent or resolve each of these issues.

- **Resource affinity.** Resource affinity can prevent you from adding more servers, or resource affinity can reduce the benefits of adding more CPUs and memory. Resource affinity occurs when code needs a specific thread, CPU, component instance, or server.
- **Excessive allocations.** Applications that allocate memory excessively on a per-request basis consume memory and create additional work for garbage collection. The additional garbage collection work increases CPU utilization. These excessive allocations may be caused by temporary allocations. For example, the excessive allocations may be caused by excessive string concatenation that uses the += operator in a tight loop.
- **Failure to share expensive resources.** Failing to call the **Dispose** or **Close** method to release expensive resources, such as database connections, may lead to resource shortages. Closing or disposing resources permits the resources to be reused more efficiently.

- **Blocking operations.** The single thread that handles an ASP.NET request is blocked from servicing additional user requests while the thread is waiting for a downstream call to return. Calls to long-running stored procedures and remote objects may block a thread for a significant amount of time.
- **Misusing threads.** Creating threads for each request incurs thread initialization costs that can be avoided. Also, using single-threaded apartment (STA) COM objects incorrectly may cause multiple requests to queue up. Multiple requests in the queue slow performance and create scalability issues.
- **Making late-bound calls.** Late-bound calls require extra instructions at runtime to identify and load the code to be run. Whether the target code is managed or unmanaged, you should avoid these extra instructions.
- **Misusing COM interop.** COM interop is generally very efficient, although many factors affect its performance. These factors include the size and type of the parameters that you pass across the managed/unmanaged boundary and crossing apartment boundaries. Crossing apartment boundaries may require expensive thread switches.
- **Large pages.** Page size is affected by the number and the types of controls on the page. Page size is also affected by the data and images that you use to render the page. The more data you send over the network, the more bandwidth you consume. When you consume high levels of bandwidth, you are more likely to create a bottleneck.
- **Failure to use data caching appropriately.** Failure to cache static data, caching too much data so that the items get flushed out, caching user data instead of application-wide data, and caching infrequently used items may limit your system's performance and scalability.
- **Failure to use output caching appropriately.** If you do not use output caching or if you use it incorrectly, you can add avoidable strain to your Web server.
- **Inefficient rendering.** Interspersing HTML and server code, performing unnecessary initialization code on page post back, and late-bound data binding may all cause significant rendering overhead. This may decrease the perceived and true page performance.

By following best practice design guidelines, you significantly increase your chances of creating a high-performance Web application. Consider the following design guidelines:

- **Consider security and performance.**
- **Partition your application logically.**
- **Evaluate affinity.**
- **Reduce round trips.**
- **Avoid blocking on long-running tasks.**
- **Use caching.**
- **Avoid unnecessary exceptions.**

Consider Security and Performance

Your choice of authentication scheme can affect the performance and scalability of your application. You need to consider the following issues:

- **Identities.** Consider the identities you are using and the way that you flow identity through your application. To access downstream resources, you can use the ASP.NET process identity or another specific service identity. Or, you can enable impersonation and flow the identity of the original caller. If you connect to Microsoft SQL Server™, you can also use SQL authentication. However, SQL authentication requires you to store credentials in the database connection string. Storing credentials in the database connection string is not recommended from a security perspective. When you connect to a shared resource, such as a database, by using a single identity, you benefit from connection pooling. Connection pooling significantly increases scalability. If you flow the identity of the original caller by using impersonation, you cannot benefit from efficient connection pooling, and you have to configure access control for multiple individual user accounts. For these reasons, it is best to use a single trusted identity to connect to downstream databases.
- **Managing credentials.** Consider the way that you manage credentials. You have to decide if your application stores and verifies credentials in a database, or if you want to use an authentication mechanism provided by the operating system where credentials are stored for you in the Active Directory® directory service.

You should also determine the number of concurrent users that your application can support and determine the number of users that your credential store (database or Active Directory) can handle. You should perform capacity planning for your application to determine if the system can handle the anticipated load.

- **Protecting credentials.** Your decision to encrypt and decrypt credentials when they are sent over the network costs additional processing cycles. If you use authentication schemes such as Windows® Forms authentication or SQL authentication, credentials flow in clear text and can be accessed by network eavesdroppers. In these cases, how important is it for you to protect them as they are passed across the network? Decide if you can choose authentication schemes that are provided by the operating system, such as NTLM or the Kerberos protocol, where credentials are not sent over the network to avoid encryption overhead.
- **Cryptography.** If your application only needs to ensure that information is not tampered with during transit, you can use keyed hashing. Encryption is not required in this case, and it is relatively expensive compared to hashing. If you need to hide the data that you send over the network, you require encryption and probably keyed hashing to ensure data validity. When both parties can share the keys, using symmetric encryption provides improved performance in comparison to asymmetric encryption. Although larger key sizes provide greater encryption strength, performance is slower relative to smaller key sizes. You must consider this type of performance and balance the larger key sizes against security tradeoffs at design time.

Partition Your Application Logically

Use layering to logically partition your application logic into presentation, business, and data access layers. This helps you create maintainable code, but it also permits you to monitor and optimize the performance of each layer separately. A clear logical separation also offers more choices for scaling your application. Try to reduce the amount of code in your code-behind files to improve maintenance and scalability.

Do not confuse logical partitioning with physical deployment. A logical separation enables you to decide whether to locate presentation and business logic on the same server and clone the logic across servers in a Web farm, or to decide to install the logic on servers that are physically separate. The key point to remember is that remote calls incur a latency cost, and that latency increases as the distance between the layers increases.

For example, in-process calls are the quickest calls, followed by cross-process calls on the same computer, followed by remote network calls. If possible, try to keep the logical partitions close to each other. For optimum performance you should place your business and data access logic in the Bin directory of your application on the Web server.

For more information about these and other deployment issues, see "Deployment Considerations" later in this chapter.

Evaluate Affinity

Affinity can improve performance. However, affinity may affect your ability to scale. Common coding practices that introduce resource affinity include the following:

- **Using in-process session state.** To avoid server affinity, maintain ASP.NET session state out of process in a SQL Server database or use the out-of-process state service running on a remote machine. Alternatively, design a stateless application, or store state on the client and pass it with each request.
- **Using computer-specific encryption keys.** Using computer-specific encryption keys to encrypt data in a database prevents your application from working in a Web farm because common encrypted data needs to be accessed by multiple Web servers. A better approach is to use computer-specific keys to encrypt a shared symmetric key. You use the shared symmetric key to store encrypted data in the database.

Reduce Round Trips

Use the following techniques and features in ASP.NET to minimize the number of round trips between a Web server and a browser, and between a Web server and a downstream system:

- **HttpResponse.IsClientConnected.** Consider using the **HttpResponse.IsClientConnected** property to verify if the client is still connected before processing a request and performing expensive server-side operations. However, this call may need to go out of process on IIS 5.0 and can be very expensive. If you use it, measure whether it actually benefits your scenario.
- **Caching.** If your application is fetching, transforming, and rendering data that is static or nearly static, you can avoid redundant hits by using caching.
- **Output buffering.** Reduce roundtrips when possible by buffering your output. This approach batches work on the server and avoids chatty communication with the client. The downside is that the client does not see any rendering of the page until it is complete. You can use the **Response.Flush** method. This method sends output up to that point to the client. Note that clients that connect over slow networks where buffering is turned off, affect the response time of your server. The response time of your server is affected because your server needs to wait

for acknowledgements from the client. The acknowledgements from the client occur after the client receives all the content from the server.

- **Server. Transfer.** Where possible, use the **Server. Transfer** method instead of the **Response. Redirect** method. **Response. Redirect** sends a response header to the client that causes the client to send a new request to the redirected server by using the new URL. **Server. Transfer** avoids this level of indirection by simply making a server-side call.

You cannot always just replace **Response. Redirect** calls with **Server. Transfer** calls because **Server. Transfer** uses a new handler during the handler phase of request processing. If you need authentication and authorization checks during redirection, use **Response. Redirect** instead of **Server. Transfer** because the two mechanisms are not equivalent. When you use **Response. Redirect**, ensure you use the overloaded method that accepts a Boolean second parameter, and pass a value of **false** to ensure an internal exception is not raised.

Also note that you can only use **Server. Transfer** to transfer control to pages in the same application. To transfer to pages in other applications, you must use **Response. Redirect**.

Avoid Blocking on Long-Running Tasks

If you run long-running or blocking operations, consider using the following asynchronous mechanisms to free the Web server to process other incoming requests:

- Use asynchronous calls to invoke Web services or remote objects when there is an opportunity to perform additional parallel processing while the Web service call proceeds. Where possible, avoid synchronous (blocking) calls to Web services because outgoing Web service calls are made by using threads from the ASP.NET thread pool. Blocking calls reduce the number of available threads for processing other incoming requests.

For more information, see "Avoid Asynchronous Calls Unless You Have Additional Parallel Work" later in this chapter.

- Consider using the **One-way** attribute on Web methods or remote object methods if you do not need a response. This "fire and forget" model allows the Web server to make the call and continue processing immediately. This choice may be an appropriate design choice for some scenarios.
- Queue work, and then poll for completion from the client. This permits the Web server to invoke code and then let the Web client poll the server to confirm that the work is complete.

Use Caching

A well-designed caching strategy is probably the single most important performance-related design consideration. ASP.NET caching features include output caching, partial page caching, and the cache API. Design your application to take advantage of these features.

Caching can be used to reduce the cost of data access and rendering output. Knowing how your pages use or render data enables you to design efficient caching strategies. Caching is particularly useful when your Web application constantly relies on data from remote resources such as databases, Web services, remote application servers, and other remote resources. Applications that are database intensive may benefit from caching by reducing the load on the database and by increasing the throughput of the application. As a general rule, if caching is cheaper than the equivalent processing, you should use caching. Consider the following when you design for caching:

- **Identify data or output that is expensive to create or retrieve.** Caching data or output that is expensive to create or retrieve can reduce the costs of obtaining the data. Caching the data reduces the load on your database server.
- **Evaluate the volatility.** For caching to be effective, the data or output should be static or infrequently modified. Lists of countries, states, or zip codes are some simple examples of the type of data that you might want to cache. Data or output that changes frequently is usually less suited to caching but can be manageable, depending upon the need. Caching user data is typically only recommended when you use specialized caches, such as the ASP.NET session state store.
- **Evaluate the frequency of use.** Caching data or output that is frequently used can provide significant performance and scalability benefits. You can obtain performance and scalability benefits when you cache static or frequently modified data and output alike. For example, frequently used, expensive data that is modified on a periodic basis may still provide large performance and scalability improvements when managed correctly. If the data is used more often than it is updated, the data is a candidate for caching.
- **Separate volatile data from nonvolatile data.** Design user controls to encapsulate static content such as navigational aids or help systems, and keep them separate from more volatile data. This permits them to be cached. Caching this data decreases the load on your server.
- **Choose the right caching mechanism.** There are many different ways to cache data. Depending on your scenario, some are better than others. User-specific data is typically stored in the **Session** object. Static pages and some types of dynamic pages such as non-personalized pages that are served to large user sets can be cached by using the ASP.NET output cache and response caching. Static content in pages can be cached by using a combination of the output cache and user controls. The ASP.NET caching features provide a built-in mechanism to update the cache. Application state, session state, and other caching means do not provide a built-in mechanism to update the cache.

Avoid Unnecessary Exceptions

Exceptions add significant overhead to your application. Do not use exceptions to control logic flow, and design your code to avoid exceptions where possible. For example, validate user input, and check for known conditions that can cause exceptions. Also, design your code to fail early to avoid unnecessary processing.

If your application does not handle an exception, it propagates up the stack and is ultimately handled by the ASP.NET exception handler. When you design your exception handling strategy, consider the following:

- **Design code to avoid exceptions.** Validate user input and check for known conditions that can cause exceptions. Design code to avoid exceptions.
- **Avoid using exceptions to control logic flow.** Avoid using exception management to control regular application logic flow.
- **Avoid relying on global handlers for all exceptions.** Exceptions cause the runtime to manipulate and walk the stack. The further the runtime traverses the stack searching for an exception handler, the more expensive the exception is to process.
- **Catch and handle exceptions close to where they occur.** When possible, catch and handle exceptions close to where they occur. This avoids excessive and expensive stack traversal and manipulation.
- **Do not catch exceptions you cannot handle.** If your code cannot handle an exception, use a **try/finally** block to ensure that you close resources, regardless of whether an exception occurs. When you use a **try/finally** block, your resources are cleaned up in the **finally** block if an exception occurs, and the exception is permitted to propagate up to an appropriate handler.
- **Fail early to avoid expensive work.** Design your code to avoid expensive or long-running work if a dependent task fails.
- **Log exception details for administrators.** Implement an exception logging mechanism that captures detailed information about exceptions so that administrators and developers can identify and remedy any issues.
- **Avoid showing too much exception detail to users.** Avoid displaying detailed exception information to users, to help maintain security and to reduce the amount of data that is sent to the client.

Resource Management

Poor resource management from pages and controls is one of the primary causes of poor Web application performance. Poor resource management can place excessive loads on CPUs and can consume vast amounts of memory. When CPU or memory thresholds are exceeded, applications might be recycled or blocked until the load on the server is lower. For more information, see "Resource Management" in Chapter 3, "Design Guidelines for Application Performance." Use the following guidelines to help you manage your resources efficiently:

- **Pool resources.**
- **Explicitly call Dispose or Close on resources you open.**
- **Do not cache or block on pooled resources.**
- **Know your application allocation pattern.**
- **Obtain resources late and release them early.**
- **Avoid per-request impersonation.**

Pool Resources

ADO.NET provides built-in database connection pooling that is fully automatic and requires no specific coding. Make sure that you use the same connection string for every request to access the database.

Make sure you release pooled resources so that they can be returned to the pool as soon as possible. Do not cache pooled resources or make lengthy blocking calls while you own the pooled resource, because this means that other clients cannot use the resource in the meantime. Also, avoid holding objects across multiple requests.

Explicitly Call Dispose or Close on Resources You Open

If you use objects that implement the **IDisposable** interface, make sure you call the **Dispose** method of the object or the **Close** method if one is provided. Failing to call **Close** or **Dispose** prolongs the life of the object in memory long after the client stops using it. This defers the cleanup and can contribute to memory pressure. Database connection and files are examples of shared resources that should be explicitly closed. The **finally** clause of the **try/finally** block is a good place to ensure that the **Close** or **Dispose** method of the object is called. This technique is shown in the following Visual Basic® .NET code fragment.

```
Try
    conn.Open()
...Finally
    If Not(conn Is Nothing) Then
        conn.Close()
    End If
End Try
```

In Visual C#®, you can wrap resources that should be disposed, by using a **using** block. When the **using** block completes, **Dispose** is called on the object listed in the brackets on the **using** statement. The following code fragment shows how you can wrap resources that should be disposed by using a **using** block.

```
SqlConnection conn = new SqlConnection(connString);
using (conn)
{
    conn.Open();
    . . .
} // Di
```

The efficiency of your ASP.NET page and code-behind page logic plays a large part in determining the overall performance of your Web application. The following guidelines relate to the development of individual .aspx and .ascx Web page files.

- **Trim your page size.**
- **Enable buffering.**
- **Use Page.IsPostBack to minimize redundant processing.**
- **Partition page content to improve caching efficiency and reduce rendering.**
- **Ensure pages are batch compiled.**
- **Ensure debug is set to false.**
- **Optimize expensive loops.**
- **Consider using Server.Transfer instead of Response.Redirect.**

- Use client-side validation.

Trim Your Page Size

Processing large page sizes increases the load on the CPU, increases the consumption of network bandwidth, and increases the response times for clients. Avoid designing and developing large pages that accomplish multiple tasks, particularly where only a few tasks are normally executed for each request. Where possible logically partition your pages.

To trim your page size, you can do one or all of the following:

- Use script includes for any static scripts in your page to enable the client to cache these scripts for subsequent requests. The following script element shows how to do this.
- `<script language=jscript src="scripts\myscript.js">`
- Remove characters such as tabs and spaces that create white space before you send a response to the client. Removing white spaces can dramatically reduce the size of your pages. The following sample table contains white spaces.

```
// with white space
<table>
  <tr>
    <td>hello</td>
    <td>world</td>
  </tr>
</table>
```

The following sample table does not contain white spaces.

```
// without white space
<table>
<tr><td>hello</td><td>world</td></tr>
</table>
```

Save these two tables in separate text files by using Notepad, and then view the size of each file. The second table saves several bytes simply by removing the white space. If you had a table with 1,000 rows, you could reduce the response time by just removing the white spaces. In intranet scenarios, removing white space may not represent a huge saving. However, in an Internet scenario that involves slow clients, removing white space can increase response times dramatically. You can also consider HTTP compression; however, HTTP compression affects CPU utilization.

You cannot always expect to design your pages in this way. Therefore, the most effective method for removing the white space is to use an Internet Server API (ISAPI) filter or an **Http Module** object. An ISAPI filter is faster than an **Http Module**; however, the ISAPI filter is more complex to develop and increases CPU utilization. You might also consider IIS compression. IIS compression can be added by using a metabase entry.

Additionally, you can trim page size in the following ways:

- Disable view state when you do not need it. For more information, see "View State" later in this chapter.
- Limit the use of graphics, and consider using compressed graphics.
- Consider using cascading style sheets to avoid sending the same formatting directives to the client repeatedly.
- Avoid long control names; especially ones that are repeated in a **Data Grid** or **Repeater** control. Control names are used to generate unique HTML ID names. A 10-character control name can easily turn into 30 to 40 characters when it is used inside nested controls that are repeated.

Enable Buffering

Because buffering is enabled by default, ASP.NET batches work on the server and avoid chatty communication with the client. The disadvantage to this approach is that for a slow page, the client does not see any rendering of the page until it is complete. You can use **Response.Flush** to mitigate this situation because **Response.Flush** sends output up to that point to the client. Clients that connect over slow networks affect the response time of your server because your server has to wait for acknowledgements from the client to proceed. Because you sent headers with the first send, there is no chance to do it later.

If buffering is turned off, you can enable buffering by using the following methods:

- Enable buffering programmatically in a page.
- ```
// Response.Buffer is available for backwards compatibility; do not use.
```
- ```
Response.BufferOutput = true;
```
- Enable buffering at the page level by using the **@Page** directive.
- ```
<%@ Page Buffer = "true" %>
```
- Enable buffering at the application or computer level by using the **<pages>** element in the Web.config or Machine.config file.
- ```
<pages buffer="true" ...>
```

When you run your ASP.NET application by using the ASP.NET process model, it is even more important to have buffering enabled. The ASP.NET worker process first sends responses to IIS in the form of response buffers. After the ISAPI filter is running, IIS receives the response buffers. These response buffers are 31 KB in size. After IIS receives the response buffers, it then sends that actual response back to the client. With buffering disabled, instead of using the entire 31-KB buffer, ASP.NET can only send a few characters to the buffer. This causes extra CPU processing in both ASP.NET as well as in IIS. This may also cause memory consumption in the IIS process to increase dramatically.

Use Page.IsPostBack to Minimize Redundant Processing

Use the **Page.IsPostBack** property to ensure that you only perform page initialization logic when a page is first loaded and not in response to client post backs. The following code fragment shows how to use the **Page.IsPostBack** property.

```
if (Page.IsPostBack == false) {
```

```
// Initialization logic
} else {
// Client post-back logic
}
```

Partition Page Content to Improve Caching Efficiency and Reduce Rendering

Partition the content in your page to increase caching potential. Partitioning your page content enables you to make different decisions about how you retrieve, display, and cache the content. You can use user controls to segregate static content, such as navigational items, menus, advertisements, copyrights, page headers, and page footers. You should also separate dynamic content and user-specific content for maximum flexibility when you want to cache content.

Ensure Pages Are Batch Compiled

As the number of assemblies that are loaded in a process grows, the virtual address space can become fragmented. When the virtual address space is fragmented, out-of-memory conditions are more likely to occur. To prevent a large number of assemblies from loading in a process, ASP.NET tries to compile all pages that are in the same directory into a single assembly. This occurs when the first request for a page in that directory occurs. Use the following techniques to reduce the number of assemblies that are not batch compiled:

- Do not mix multiple languages in the same directory. When multiple languages such as C# or Visual Basic .NET are used in pages in the same directory, ASP.NET compiles a separate assembly for each language.
- Ensure content updates do not cause additional assemblies to be loaded. For more information, see "Deployment Considerations" later in this chapter.
- Ensure that the **debug** attribute is set to **false** at the page level and in the Web.config file, as described in the following section.

Ensure Debug Is Set to False

When **debug** is set to **true**, the following occurs:

- Pages are not batch compiled.
- Pages do not time out. When a problem occurs, such as a problem with a Web service call, the Web server may start to queue requests and stop responding.
- Additional files are generated in the Temporary ASP.NET Files folder.
- The **System.Diagnostics.DebuggableAttribute** attribute is added to generated code. This causes the CLR to track extra information about generated code, and it also disables certain optimizations.

Before you run performance tests and before you move your application into production, be sure that **debug** is set to **false** in the Web.config file and at the page level. By default, **debug** is set to **false** at the page level. If you do need to set this attribute during development time, it is recommended that you set it at the Web.config file level, as shown in the following fragment.

```
<compilation debug="false" ... />
```

The following shows how to set **debug** to **false** at the page level.

```
<%@ Page debug="false" ...
```

Optimize Expensive Loops

Expensive loops in any application can cause performance problems. To reduce the overhead that is associated with code inside loops, you should follow these recommendations:

- Avoid repetitive field or property access.
- Optimize code inside the loop.
- Copy frequently called code into the loop.
- Replace recursion with looping.
- Use **For** instead of **For Each** in performance-critical code paths.

Consider Using **Server.Transfer** Instead of **Response.Redirect**

Response.Redirect sends a metatag to the client that makes the client send a new request to the server by using the new URL. **Server.Transfer** avoids this indirection by making a server-side call. When you use **Server.Transfer**, the URL in the browser does not change, and load test tools may incorrectly report the page size because different pages are rendered for the same URL.

The **Server.Transfer**, **Response.Redirect**, and **Response.End** methods all raise **ThreadAbortException** exceptions because they internally call **Response.End**. The call to **Response.End** causes this exception. Consider using the overloaded method to pass **false** as the second parameter so that you can suppress the internal call to **Response.End**.

Use Client-Side Validation

Prevalidating data can help reduce the round trips that are required to process a user's request. In ASP.NET, you can use validation controls to implement client-side validation of user input.

Server Controls

You can use server controls to encapsulate and to reuse common functionality. Server controls provide a clean programming abstraction and are the recommended way to build ASP.NET applications. When server controls are used properly, they can improve output caching and code maintenance. The main areas you should review for performance optimizations are view state and control composition. Use the following guidelines when you develop server controls:

- **Identify the use of view state in your server controls.**
- **Use server controls where appropriate.**
- **Avoid creating deep hierarchies of controls.**

Identify the Use of View State in Your Server Controls

View state is serialized and deserialized on the server. To save CPU cycles, reduce the amount of view state that your application uses. Disable view state if you do not need it. Disable view state if you are doing at least one of the following:

- Displaying a read-only page where there is no user input
- Displaying a page that does not post back to the server
- Rebuilding server controls on each post back without checking the post back data

Use Server Controls Where Appropriate

The HTTP protocol is stateless; however, server controls provide a rich programming model that manages state between page requests by using view state. Server controls require a fixed amount of processing to establish the control and all of its child controls. This makes server controls relatively expensive compared to HTML controls or possibly static text. Scenarios where server controls are expensive include the following:

- **Large payload over low bandwidth.** The more controls that you have on a page, the higher the network payload is. Therefore, multiple controls decreases the time to last byte (TTLB) and the time to first byte (TTFB) for the response that is sent to the client. When the bandwidth between client and server is limited, as is the case when a client uses a low-speed dial-up connection, pages that carry a large view state payload can significantly affect performance.
- **View state overhead.** View state is serialized and deserialized on the server. The CPU effort is proportional to the view state size. In addition to server controls that use view state, it is easy to programmatically add any object that can be serialized to the view state property. However, adding objects to the view state adds to the overhead. Other techniques such as storing, computed data or storing several copies of common data adds unnecessary overhead.
- **Composite controls or large number of controls.** Pages that have composite controls such as **Data Grid** may increase the footprint of the view state. Pages that have a large number of server controls also may increase the footprint of the view state. Where possible, consider the alternatives that are presented later in this section.

When you do not need rich interaction, replace server controls with an inline representation of the user interface that you want to present. You might be able to replace a server control under the following conditions:

- You do not need to retain state across post backs.
- The data that appears in the control is static. For example, a label is static data.
- You do not need programmatic access to the control on the server-side.
- The control is displaying read-only data.
- The control is not needed during post back processing.

Alternatives to server controls include simple rendering, HTML elements, inline **Response.Write** calls, and raw inline angle brackets (<% %>). It is essential to balance your tradeoffs.

Avoid over optimization if the overhead is acceptable and if your application is within the limits of its performance objectives.

Avoid Creating Deep Hierarchies of Controls

Deeply nested hierarchies of controls compound the cost of creating a server control and its child controls. Deeply nested hierarchies create extra processing that could be avoided by using a different design that uses inline controls, or by using a flatter hierarchy of server controls. This is especially important when you use list controls such as **Repeater**, **Data List**, and **Data Grid** because they create additional child controls in the container.

For example, consider the following **Repeater** control.

```
<asp: repeater id=r runat=server>
  <item template>
    <asp: label runat=server><%# Container.DataItem %><br></asp: label>
  </itemtemplate>
</asp:repeater>
```

Assuming there are 50 items in the data source, if you enable tracing for the page that contains the **Repeater** control, you would see that the page actually contains more than 200 controls.

Data Binding

Data binding is another common area that often leads to performance problems if it is used inefficiently. If you use data binding, consider the following recommendations:

- **Avoid using Page.DataBind.**
- **Minimize calls to DataBinder.Eval.**

Avoid Using Page.DataBind

Calling **Page.DataBind** invokes the page-level method. The page-level method in turn calls the **DataBind** method of every control on the page that supports data binding. Instead of calling the page-level **DataBind**, call **DataBind** on specific controls. Both approaches are shown in the following examples.

The following line calls the page level **DataBind**. The page level **DataBind** in turn recursively calls **DataBind** on each control.

```
DataBind();
```

The following line calls **DataBind** on the specific control.

```
yourServerControl.DataBind();
```

Minimize Calls to `DataBinder.Eval`

The **`DataBinder.Eval`** method uses reflection to evaluate the arguments that are passed in and to return the results. If you have a table that has 100 rows and 10 columns, you call **`DataBinder.Eval`** 1,000 times if you use **`DataBinder.Eval`** on each column. Your choice to use **`DataBinder.Eval`** is multiplied 1,000 times in this scenario. Limiting the use of **`DataBinder.Eval`** during data binding operations significantly improves page performance. Consider the following **Item Template** element within a **Repeater** control using **`DataBinder.Eval`**.

```
<ItemTemplate>
  <tr>
    <td><%# DataBinder.Eval(Container.DataItem, "field1") %></td>
    <td><%# DataBinder.Eval(Container.DataItem, "field2") %></td>
  </tr>
</ItemTemplate>
```

There are alternatives to using **`DataBinder.Eval`** in this scenario. The alternatives include the following:

- **Use explicit casting.** Using explicit casting offers better performance by avoiding the cost of reflection. Cast the **`Container.DataItem`** as a **`DataRowView`**.

```
<ItemTemplate>
  <tr>
    <td><%# ((DataRowView)Container.DataItem) ["field1"] %></td>
    <td><%# ((DataRowView)Container.DataItem) ["field2"] %></td>
  </tr>
</ItemTemplate>
```

You can gain even better performance with explicit casting if you use a **`DataReader`** to bind your control and use the specialized methods to retrieve your data. Cast the **`Container.DataItem`** as a **`DbDataRecord`**.

```
<ItemTemplate>
  <tr>
    <td><%# ((DbDataRecord)Container.DataItem).GetString(0) %></td>
    <td><%# ((DbDataRecord)Container.DataItem).GetInt(1) %></td>
  </tr>
</ItemTemplate>
```

The explicit casting depends on the type of data source you are binding to; the preceding code illustrates an example.

- **Use the `ItemDataBound` event.** If the record that is being data bound contains many fields, it may be more efficient to use the **`ItemDataBound`** event. By using this event, you only perform the type conversion once. The following sample uses a **`DataSet`** object.

```
protected void Repeater_ItemDataBound(Object sender,
RepeaterItemEventArgs e)
{
```

```
• DataRowView drv = (DataRowView)e.Item.DataItem;
• Response.Write(string.Format("<td>{0}</td>", drv["field1"]));
• Response.Write(string.Format("<td>{0}</td>", drv["field2"]));
• Response.Write(string.Format("<td>{0}</td>", drv["field3"]));
• Response.Write(string.Format("<td>{0}</td>", drv["field4"]));
• }
```

Caching Explained

Caching avoids redundant work. If you use caching properly, you can avoid unnecessary database lookups and other expensive operations. You can also reduce latency.

The ASP.NET cache is a simple, scalable, in-memory caching service provided to ASP.NET applications. It provides a time-based expiration facility, and it also tracks dependencies on external files, directories, or other cache keys. It also provides a mechanism to invoke a callback function when an item expires in the cache. The cache automatically removes items based on a least recently used (LRU) algorithm, a configured memory limit, and the **CacheItemPriority** enumerated value of the items in the cache. Cached data is also lost when your application or worker process recycles.

ASP.NET provides the following three caching techniques:

- **Cache API**
- **Output caching**
- **Partial page or fragment caching**

These caching techniques are briefly summarized in the following sections.

Cache API

You should use the cache API to programmatically cache application-wide data that is shared and accessed by multiple users. The cache API is also a good place for data that you need to manipulate in some way before you present the data to the user. This includes data such as strings, arrays, collections, and data sets.

Some common scenarios where you might want to use the cache API include the following:

- **Headlines.** In most cases, headlines are not updated in real time; they are often delayed for 10 to 20 minutes. Because headlines are shared by multiple users and are updated infrequently, this makes them good candidates for the cache API.
- **Product catalogs.** Product catalogs are good candidates for the cache API because the data typically needs to be updated at specific intervals, shared across the application, and manipulated before sending the content to the client.

You should avoid using the cache API in the following circumstances:

- The data you are caching is user-specific. Consider using session state instead.
- The data is updated in real time.
- Your application is already in production, and you do not want to update the code base. In this case, consider using output caching.

The cache API permits you to insert items in the cache that have a dependency upon external conditions. Cached items are automatically removed from the cache when the external conditions change. You use this feature by using a third parameter on the **Cache.Insert** method that accepts an instance of a **CacheDependency** class. The **CacheDependency** class has eight different constructors that support various dependency scenarios. These constructors include file-based, time-based, and priority-based dependencies, together with dependencies that are based on existing dependencies.

You can also run code before serving data from the cache. For example, you might want to serve cached data for certain customers, but for others you might want to serve data that is updated in real time. You can perform this type of logic by using the **HttpCachePolicy.AddValidationCallback** method.

Output Caching

The output cache enables you to cache the contents of entire pages for a specific duration of time. It enables you to cache multiple variations of the page based on query strings, headers, and **userAgent** strings. The output cache also enables you to determine where to cache the content, for example on a proxy, server, or a client. Like the cache API, output caching enables you to save time retrieving data. Output caching also saves time rendering content. You should enable output caching on dynamically generated pages that do not contain user-specific data in scenarios where you do not need to update the view on every request.

Some common scenarios that are ideal for output caching include the following:

- **Pages that are frequently visited.** Output caching can be used to increase the overall performance of an application after the application is released to production by identifying the heavily visited pages and by enabling output caching on those specific pages if possible.
- **Reports.** Reports that only contain a low number of variations are good candidates for using output caching because you save time by not retrieving and processing the data each time the page is accessed.

Avoid using output caching in the following circumstances:

- You need programmatic access to the data on your page. Consider using the cache API instead.
- The number of page variants becomes too large.
- The page contains a mixture of static, dynamic, and user-specific data. Consider using fragment caching instead.
- The page contains content that must be refreshed with every view.

Partial Page or Fragment Caching

Partial page or fragment caching is a subset of output caching. It includes an additional attribute that allows you to cache a variation based on the properties of the user control (.aspx file.)

Fragment caching is implemented by using user controls in conjunction with the **@OutputCache** directive. Use fragment caching when caching the entire content of a page is not practical. If you have a mixture of static, dynamic, and user-specific content in your page, partition your page into separate logical regions by creating user controls. These user controls can then be cached, independent of the main page, to reduce processing time and to increase performance.

Some common scenarios that make good candidates for fragment caching include the following:

- **Navigation menus.** Navigation menus that are not user-specific are great candidates for fragment caching because menus are usually rendered with each request and are often static.
- **Headers and footers.** Because headers and footers are essentially static content that does not need to be regenerated with every request, they make good candidates for fragment caching.

You should avoid using fragment caching under the following conditions:

- The number of page variants becomes too large.
- The cached user controls contain content that must be refreshed with every view.

If your application uses the same user control on multiple pages, make the pages share the same instance by setting the **Shared** attribute of the user control **@ OutputCache** directive to **true**. This can save a significant amount of memory.

Caching Guidelines

Consider the following guidelines when you are designing a caching strategy:

- **Separate dynamic data from static data in your pages.**
- **Configure the memory limit.**
- **Cache the right data.**
- **Refresh your cache appropriately.**
- **Cache the appropriate form of data.**
- **Use output caching to cache relatively static pages.**
- **Choose the right cache location.**
- **Use VaryBy attributes for selective caching.**
- **Use kernel caching on Windows Server 2003.**

Separate Dynamic Data from Static Data in Your Pages

Partial page caching enables you to cache parts of a page by using user controls. Use user controls to partition your page. For example, consider the following simple page which contains static, dynamic, and user-specific information.

```
[main.aspx]
<html>
<body>
<table>
<tr><td colspan=3>Application Header - Welcome John Smith</td></tr>
<tr><td>Menu</td><td>Dynamic Content</td><td>Advertisements</td></tr>
<tr><td colspan=3>Application Footer</td></tr>
</table>
</html>
```

You can partition and cache this page by using the following code:

```
[main.aspx]
<%@ Register TagPrefix="app" TagName="header" src="header.ascx" %>
<%@ Register TagPrefix="app" TagName="menu" src="menu.ascx" %>
<%@ Register TagPrefix="app" TagName="advertisements"
src="advertisements.ascx" %>
<%@ Register TagPrefix="app" TagName="footer" src="footer.ascx" %>
<html>
<body>
<table>
<tr><td colspan=3><app:header runat=server /></td></tr>
<tr><td><app:menu runat=server /></td><td>Dynamic
Content</td><td><app:advertisements runat=server /></td></tr>
<tr><td colspan=3><app:footer runat=server /></td></tr>
</table>
</html>

[header.ascx]
<%@Control %>
Application Header - Welcome <% GetName() %>

[menu.ascx]
<%@Control %>
<%@ OutputCache Duration="30" VaryByParam="none" %>
Menu

[advertisements.ascx]
<%@Control %>
<%@ OutputCache Duration="30" VaryByParam="none" %>
Advertisements

[footer.ascx]
<%@Control %>
<%@ OutputCache Duration="60" VaryByParam="none" %>
Footer
```

By partitioning the content, as shown in the sample, you can cache selected portions of the page to reduce processing and rendering time.

Configure the Memory Limit

Configuring and tuning the memory limit is critical for the cache to perform optimally. The ASP.NET cache starts trimming the cache based on a LRU algorithm and the **CacheItemPriority** enumerated value assigned to the item after memory consumption is within 20 percent of the configured memory limit. If the memory limit is set too high, it is possible for the process to be recycled unexpectedly. Your application might also experience out-of-memory exceptions. If the memory limit is set too low, it could increase the amount of time spent performing garbage collections, which decreases overall performance.

Empirical testing shows that the likelihood of receiving out-of-memory exceptions increases when private bytes exceed 800 megabytes (MB). A good rule to follow when determining when to increase or decrease this number is that 800 MB is only relevant for .NET Framework 1.0. If you have .NET Framework 1.1 and if you use the /3 GB switch, you can go up to 1,800 MB.

When using the ASP.NET process model, you configure the memory limit in the Machine.config file as follows.

```
<processModel memoryLimit="50">
```

This value controls the percentage of physical memory that the worker process is allowed to consume. The process is recycled if this value is exceeded. In the previous sample, if there are 2 gigabytes (GB) of RAM on your server, the process recycles after the total available physical RAM falls below 50 percent of the RAM; in this case 1 GB. In other words, the process recycles if the memory used by the worker process goes beyond 1 GB. You monitor the worker process memory by using the **process** performance counter object and the **private bytes** counter.

Cache the Right Data

It is important to cache the right data. If you cache the wrong data, you may adversely affect performance.

Cache application-wide data and data that is used by multiple users. Cache static data and dynamic data that is expensive to create or retrieve. Data that is expensive to retrieve and that is modified on a periodic basis can still provide performance and scalability improvements when managed properly. Caching data even for a few seconds can make a big difference to high volume sites. Datasets or custom classes that use optimized serialization for data binding are also good candidates for caching. If the data is used more often than it is updated, it is also a candidate for caching.

Do not cache expensive resources that are shared, such as database connections, because this creates contention. Avoid storing **DataReader** objects in the cache because these objects keep the underlying connections open. It is better to pool these resources. Do not cache per-user data

that spans requests — use session state for that. If you need to store and to pass request-specific data for the life of the request instead of repeatedly accessing the database for the same request, consider storing the data in the **HttpContext.Current.Cache** object.

Refresh Your Cache Appropriately

Just because your data updates every ten minutes does not mean that your cache needs to be updated every ten minutes. Determine how frequently you have to update the data to meet your service level agreements. Avoid repopulating caches for data that changes frequently. If your data changes frequently, that data may not be a good candidate for caching.

Cache the Appropriate Form of the Data

If you want to cache rendered output, you should consider using output caching or fragment caching. If the rendered output is used elsewhere in the application, use the cache API to store the rendered output. If you need to manipulate the data, then cache the data by using the cache API. For example, if you need the data to be bound to a combo box, convert the retrieved data to an **ArrayList** object before you cache it.

Use Output Caching to Cache Relatively Static Pages

If your page is relatively static across multiple user requests, consider using page output caching to cache the entire page for a specified duration. You specify the duration based on the nature of the data on the page. A dynamic page does not always have to be rebuilt for every request just because it is a dynamic page. For example, you might be able to cache Web-based reports that are expensive to generate for a defined period. Caching dynamic pages for even a minute or two can increase performance drastically on high volume pages.

If you need to remove an item from the cache instead of waiting until the item expires, you can use the **HttpResponse.RemoveOutputCacheItem** method. This method accepts an absolute path to the page that you want to remove as shown in the following code fragment.

```
HttpResponse.RemoveOutputCacheItem("/Test/Test.aspx");
```

The caveat here is that this is specific to a server, because the cache is not shared across a Web farm. Also, it cannot be used from a user control.

Choose the Right Cache Location

The **@OutputCache** directive allows you to determine the cache location of the page by using the **Location** attribute. The **Location** attribute provides the following values:

- **Any.** This is the default value. The output cache can be located on the browser client where the request originated, on a proxy server, or any other server that is participating in the request or on the server where the request is processed.
- **Client.** The output cache is located on the browser client where the request originated.

- **DownStream.** The output cache can be stored in any HTTP 1.1 cache-capable device except for the origin server. This includes proxy servers and the client that made the request.
- **None.** The output cache is disabled for the requested page.
- **Server.** The output cache is located on the Web server where the request was processed.
- **ServerAndClient.** The output cache can be stored only at the origin server or at the requesting client. Proxy servers cannot cache the response.

Unless you know for certain that your clients or your proxy server will cache responses, it is best to keep the **Location** attribute set to **Any**, **Server**, or **ServerAndClient**. Otherwise, if there is not a downstream cache available, the attribute effectively negates the benefits of output caching.

Use VaryBy Attributes for Selective Caching

The **VaryBy** attributes allow you to cache different versions of the same page. ASP.NET provides four **VaryBy** attributes:

- **VaryByParam.** Different versions of the page are stored based on the query string values.
- **VaryByHeader.** Different versions of the page are stored based on the specified header values.
- **VaryByCustom.** Different versions of the page are stored based on browser type and major version. Additionally, you can extend output caching by defining custom strings.
- **VaryByControl.** Different versions of the page are stored based on the property value of a user control. This only applies to user controls.

The **VaryBy** attribute determines the data that is cached. The following sample shows how to use the **VaryBy** attribute.

```
<%@ OutputCache Duration="30" VaryByParam="a" %>
```

The setting shown in the previous sample would make the following pages have the same cached version:

- <http://localhost/cache.aspx?a=1>
- <http://localhost/cache.aspx?a=1&b=1>
- <http://localhost/cache.aspx?a=1&b=2>

If you add *b* to the **VaryByParam** attribute, you would have three separate versions of the page rather than one version. It is important for you to be aware of the number of variations of the cached page that could be cached. If you have two variables (*a* and *b*), and *a* has 5 different combinations, and *b* has 10 different combinations, you can calculate the total number of cached pages that could exist by using the following formula:

$$(MAX a \times MAX b) + (MAX a + MAX b) = 65 \text{ total variations}$$

When you make the decision to use a **VaryBy** attribute, make sure that there are a finite number of variations because each variation increases the memory consumption on the Web server.

State Management

Web applications present specific challenges for state management. This is especially true for Web applications that are deployed in Web farms. The choices that you make regarding where and how state is stored have a significant impact on the performance and scalability of your application. There are several different types of state:

- **Application state.** Application state is used for storing application-wide state for all clients. Using application state affects scalability because it causes server affinity. In a Web scenario, if you modify application state, there is no mechanism to replicate the changes across servers. Therefore, if a subsequent request from the same user goes to another server, the change is not available. You store data in application state by using a key/value pair, as shown in the following sample.

```
Application["YourGlobalState"] = somevalue;
```
- **Session state.** Session state is used for storing per-user state on the server. The state information is tracked by using a session cookie or a mangled URL. ASP.NET session state scales across Web servers in a farm.
- **View state.** View state is used for storing per-page state information. The state flows with every HTTP POST request and response.
- **Alternatives.** Other techniques for state management include client cookies, query strings, and hidden form fields.

Guidelines that are specific to application state, session state, and view state are included in later sections. The following are guidelines that address the broad issues that concern state management in general:

- **Store simple state on the client where possible.**
- **Consider serialization costs.**

Store Simple State on the Client Where Possible

Use cookies, query strings, and hidden controls for storing lightweight, user-specific state that is not sensitive such as personalization data. Do not use them to store security-sensitive information because the information can be easily read or manipulated.

- **Client cookies.** Client cookies are created on the server, and they are sent and stored on the client browser. They are domain specific and are not completely secure. All subsequent requests from a browser include the cookies, which the server code can inspect and modify. The maximum amount of data that you can put in cookie is 4 KB.
- **Query strings.** Query strings are the data that is appended to a URL. The data is clear text and there is a limit on the overall string length. The data can easily be manipulated by the user. Therefore, do not retrieve and display sensitive data based on query parameters without using authentication or validation. For anonymous Web sites, this is less of an issue.
- **Hidden controls.** Hidden controls on the page store state information that is sent back and forth in requests and responses.

Application State

Application state is used to store application-wide static information. ASP.NET includes application state primarily for compatibility with classic Active Server Pages (ASP) technology so that it is easier to migrate existing applications to ASP.NET.

If you use application state, use the following guidelines to ensure your application runs optimally:

- **Use static properties instead of the Application object to store application state.**
- **Use application state to share static, read-only data.**
- **Do not store STA COM objects in application state.**

Use Static Properties Instead of the Application Object to Store Application State

You should store data in static members of the application class instead of in the **Application** object. This increases performance because you can access a static variable faster than you can access an item in the **Application** dictionary. The following is a simplified example.

```
<%
private static string[] _states[];
private static object _lock = new object();
public static string[] States
{
    get {return _states;}
}
public static void PopulateStates()
{
    //ensure this is thread safe
    if(_states == null)
    {
        lock(_lock)
        {
            //populate the states...
        }
    }
}
public void Application_OnStart(object sender, EventArgs e)
{
    PopulateStates();
}
%>
```

Use Application State to Share Static, Read-Only Data

Application state is application-wide and specific to a server. Even though you can store read-write data, it is advisable to only store read-only data to avoid server affinity. Consider using the **Cache** object. The **Cache** object is a better alternative for read-only data.

Do Not Store STA COM Objects in Application State

Storing STA COM objects in application state bottlenecks your application because the application uses a single thread of execution when it accesses the component. Avoid storing STA COM objects in application state.

Session State

If you need session state in ASP.NET, there are three session state modes that you can choose from. Each mode offers varying degrees of performance and scalability as described in the following list:

- **InProc.** The in-process store provides the fastest access to session state. There are no serialization or marshaling costs involved because state is maintained within the managed memory of the ASP.NET process. The ASP.NET process is the Aspnet_wp.exe file on Windows 2000 Server, and the W3wp.exe file on Windows Server 2003. When the process recycles, the state data is lost, although you can disable process recycling in IIS 6 if process recycling affects your application. The in-process store limits application scalability because you cannot use it in conjunction with multiple worker processes; for example, it prevents Web farm or Web garden deployment. Also, high numbers of large or concurrent sessions can cause your application to run out of memory.
- **StateServer.** The session state service, a Microsoft Win32® service, can be installed on your local Web server or on a remote server that is accessible by all Web servers in a Web farm. This approach scales well, but performance is reduced in comparison to the in-process provider because of the additional serialization and marshaling that is required to transfer the state to and from the state store.
- **SQL Server.** Microsoft SQL Server provides a highly scalable and easily available solution. SQL Server is a solution that is well-suited to large amounts of session state. The serialization and marshaling costs are the same as the costs for the session state service, although overall performance is slightly lower. SQL Server provides clustering for failover, although this is not supported in the default configuration for session state. To enable clustering for failover, you have to apply configuration changes, and the session data must be stored in a non temporary table.

For more information, see Knowledge Base article 323262, "INFO: ASP.NET Session State with SqlServer Mode in a Failover Cluster," at <http://support.microsoft.com/default.aspx?scid=kb;en-us;323262>.

Choosing a State Store

The in-process state store provides excellent performance and scales well. However, most high volume Web applications run in a Web farm. To be able to scale out, you need to choose between the session state service and the SQL Server state store. With either of these choices, you have to understand the associated impact of network latency and serialization, and you have to measure them to ensure that your application meets its performance objectives. Use the following information to help choose a state store:

- **Single Web server.** Use the in-process state store when you have a single Web server, when you want optimum session state performance, and when you have a reasonable and limited number of concurrent sessions. Use the session state service running on the local Web server when your sessions are expensive to rebuild and when you require durability in the event of an ASP.NET restart. Use the SQL Server state store when reliability is your primary concern.
- **Web farm.** Avoid the in-process option, and avoid running the session state service on the local Web server. These cause server affinity. You can use Internet Protocol (IP) affinity to ensure that the same server handles subsequent requests from the same client, but Internet service providers (ISP) that use a reverse proxy cause problems for this approach. Use a remote session state service or use SQL Server for Web farm scenarios.
- **StateServer versus SQLServer.** Use a remote state service, if you do not have a SQL Server database. Use SQL Server for enterprise applications or high volume Web applications. If your remote state service and your Web server are separated by a firewall, then you need to open a port. The default port is port 42424. You can change the port in the following registry key:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\aspnet_state\Parameters.

To ensure optimized session state performance, follow these guidelines:

- **Prefer basic types to reduce serialization costs.**
- **Disable session state if you do not use it.**
- **Avoid storing STA COM objects in session state.**
- **Use the ReadOnly attribute when you can.**

Prefer Basic Types to Reduce Serialization Costs

You incur serialization overhead if you use the **StateServer** or the **SQLServer** out-of-process state stores. The simpler the object graph, the faster it should serialize. To minimize serialization costs, use basic types such as **Int**, **Byte**, **Decimal**, **String**, **DateTime**, **TimeSpan**, **Guid**, **IntPtr**, and **UIntPtr**. ASP.NET uses an optimized internal serialization method to serialize basic types. Complex types are serialized using a relatively slow **BinaryFormatter** object. For complex types, you can use the **Serializable** attribute, or you can implement the **ISerializable** interface. Using this interface provides you with more precise control and may speed up serialization.

Minimize what you serialize. Disable serialization when you do not use it, and mark specific fields from a serializable class that you want to exclude with the **NonSerialized** attribute. Alternatively, control the serialization process by using the **ISerializable** interface.

Note You should only implement the **ISerializable** interface as a last resort. New formatters provided by future versions of the .NET Framework and improvements to the framework provided serialization will not be utilized once you take this approach. Prefer the **NonSerialized** attribute.

Disable Session State If You Do Not Use It

If you do not use session state, disable session state to eliminate redundant session processing performed by ASP.NET. You might not use session state because you store simple state on the client and then pass it to the server for each request. You can disable session state for all applications on the server, for specific applications, or for individual pages, as described in the following list:

- To disable session state for all applications on your server, use the following element in the Machine.config file.

```
<sessionState mode='Off' />
```

You can also remove the session state module from **<httpModules>** to completely remove session processing overhead.

- To disable session state for a specific application, use the following element in the Web.config file of the application.

```
<sessionState mode='Off' />
```

- To disable session state for a specific Web page, use the following page setting.

```
<%@ Page EnableSessionState="false" . . .%>
```

Avoid Storing STA COM Objects in Session State

Storing STA COM objects in session state causes thread affinity. Thread affinity severely affects performance and scalability. If you do use STA COM objects in session state, be sure to set the **AspCompat** attribute of the **@ Page** directive.

View State

View state is used primarily by server controls to retain state only on pages that post data back to themselves. The information is passed to the client and read back in a specific hidden variable called **_VIEWSTATE**. ASP.NET makes it easy to store any types that are serializable in view state. However, this capability can easily be misused and performance reduced. View state is an unnecessary overhead for pages that do not need it. As the view state grows larger, it affects performance in the following ways:

- Increased CPU cycles are required to serialize and to deserialize the view state.
- Pages take longer to download because they are larger.
- Very large view state can impact the efficiency of garbage collection.

Transmitting a huge amount of view state can significantly affect application performance. The change in performance becomes more marked when your Web clients use slow, dial-up connections. Consider testing for different bandwidth conditions when you work with view state. Optimize the way your application uses view state by following these recommendations:

- **Disable view state if you do not need it.**

- **Minimize the number of objects you store in view state.**
- **Determine the size of your view state.**

Disable View State If You Do Not Need It

View state is turned on in ASP.NET by default. Disable view state if you do not need it. For example, you might not need view state because your page is output-only or because you explicitly reload data for each request. You do not need view state when the following conditions are true:

- **Your page does not post back.** If the page does not post information back to itself, if the page is only used for output, and if the page does not rely on response processing, you do not need view state.
- **You do not handle server control events.** If your server controls do not handle events, and if your server controls have no dynamic or data bound property values, or they are set in code on every request, you do not need view state.
- **You repopulate controls with every page refresh.** If you ignore old data, and if you repopulate the server control each time the page is refreshed, you do not need view state.

There are several ways to disable view state at various levels:

- To disable view state for all applications on a Web server, configure the **<pages>** element in the Machine.config file as follows.

```
<pages enableViewState="false" />
```

This approach allows you to selectively enable view state just for those pages that need it by using the **EnableViewState** attribute of the **@ Page** directive.

- To disable view state for a single page, use the **@ Page** directive as follows.
- **<%@ Page EnableViewState="false" %>**
- To disable view state for a single control on a page, set the **EnableViewState** property of the control to **false**, as shown in the following code fragment.

```
//programmatically
yourControl.EnableViewState = false;
//something
<asp:datagrid EnableViewState="false" runat= "server" />
```

Minimize the Number of Objects You Store In View State

As you increase the number of objects you put into view state, the size of your view state dictionary grows, and the processing time required to serialize and to deserialize the objects increases. Use the following guidelines when you put objects into view state:

- View state is optimized for serializing basic types such as strings, integers, and Booleans, and objects such as arrays, **ArrayLists**, and **Hashtables** if they contain these basic types. When you want to store a type which is not listed previously, ASP.NET internally tries to use the associated type converter. If it cannot find one, it uses the relatively expensive binary serializer.

- The size of the object is directly proportional to the size of the view state. Avoid storing large objects.

Determine the Size of Your View State

By enabling tracing for the page, you can monitor the view state size for each control. The view state size for each control appears in the leftmost column in the control tree section of the trace output. Use this information as a guide to determine if there are any controls that you can reduce the amount of view state for or if there are controls that you can disable view state for.

tring Management

When you build output, you often need to concatenate strings. This is an expensive operation because it requires temporary memory allocation and subsequent collection. As a result, you should minimize the amount of string concatenation that you perform. There are three common ways to concatenate strings in your pages to render data:

- **Using the += operator.** Use the += operator when the number of appends is known.
- **StringBuilder.** Use the **StringBuilder** object when the number of appends is unknown. Treat **StringBuffer** as a reusable buffer.
- **Response.Write <% %>.** Use the **Response.Write** method. It is one of the fastest ways to return output back to the browser.

The most effective way to determine the option to choose is to measure the performance of each option. If your application relies heavily on temporary buffers, consider implementing a reusable buffer pool of character arrays or byte arrays.

Use the following guidelines when you are managing your strings:

- **Use Response.Write for formatting output.**
- **Use StringBuilder for temporary buffers.**
- **Use HtmlTextWriter when building custom controls.**

Use Response.Write for Formatting Output

Where possible, avoid using loops to concatenate strings for formatting page layout. Consider using **Response.Write** instead. This approach writes output to the ASP.NET response buffers. When you are looping through datasets or XML documents, using **Response.Write** is a highly efficient approach. It is more efficient than concatenating the content by using the += operator before writing the content back to the client. **Response.Write** internally appends strings to a reusable buffer so that it does not suffer the performance overhead of allocating memory, in addition to cleaning that memory up.

Use StringBuilder for Temporary Buffers

In many cases it is not feasible to use **Response.Write**. For example, you might need to create strings to write to a log file or to build XML documents. In these situations, use a **StringBuilder** object as a temporary buffer to hold your data. Measure the performance of your scenario by trying various initial capacity settings for the **StringBuilder** object.

Exception Management

Exceptions are expensive. By knowing the causes of exceptions, and by writing code that avoids exceptions and that handles exceptions efficiently, you can significantly improve the performance and scalability of your application. When you design and implement exception handling, consider the following guidelines to ensure optimum performance:

- **Implement a Global.asax error handler.**
- **Monitor application exceptions.**
- **Use try/finally on disposable resources.**
- **Write code that avoids exceptions.**
- **Set timeouts aggressively.**

Implement a Global.asax Error Handler

The first step in managing exceptions is to implement a global error handler in the Global.asax file or in the code-behind file. Implementing a global error handler traps all unhandled exceptions in your application. Inside the handler, you should, at a minimum, log the following information to a data store such as a database, the Windows event log, or a log file:

- The page that the error occurred on
- Call stack information
- The exception name and message

In your Global.asax file or your code-behind page, use the **Application_Error** event to handle your error logic, as shown in the following code sample:

```
public void Application_Error(object s, EventArgs ev)
{
    StringBuilder message = new StringBuilder();
    if (Server != null) {
        Exception e;
        for (e = Server.GetLastError(); e != null; e = e.InnerException)
        {
            message.AppendFormat("{0}: {1}{2}",
                e.GetType().FullName,
                e.Message,
                e.StackTrace);
        }
        //Log the exception and inner exception information.
    }
}
```

```
}
```

Monitor Application Exceptions

To reduce the number of exceptions occurring in your application, you need to effectively monitor your application for exceptions. You can do the following:

- If you have implemented exception handling code, review your exception logs periodically.
- Monitor the **# of Exceps Thrown / sec** counter under the **.NET CLR Exceptions** Performance Monitor object. This value should be less than 5 percent of your average requests per second.

Use Try/Finally on Disposable Resources

To guarantee resources are cleaned up when an exception occurs, use a **try/finally** block. Close the resources in the **finally** clause. Using a **try/finally** block ensures that resources are disposed even if an exception occurs. The following code fragment demonstrates this.

```
try
{
    conn.Open();
    ...
}
finally
{
    if (null != conn)
        conn.close;
}
```

Write Code That Avoids Exceptions

The following is a list of common techniques you can use to avoid exceptions:

- **Check for null values.** If it is possible for an object to be **null**, check to make sure it is not **null**, rather than throwing an exception. This commonly occurs when you retrieve items from view state, session state, application state, or cache objects as well as query string and form field variables. For example, do not use the following code to access session state information.

```
• try {
•     loginid = Session["loginid"].ToString();
• }
• catch(Exception ex) {
•     Response.Redirect("login.aspx", false);
• }
```

Instead, use the following code to access session state information.

```
if (Session["loginid"] != null)
    loginid = Session["loginid"].ToString();
else
```

```
Response.Redirect("login.aspx", false);
```

- **Do not use exceptions to control logic.** Exceptions are just that — exceptions. A database connection that fails to open is an exception. A user who mistypes his password is simply a condition that needs to be handled. For example, consider the following function prototype used to log in a user.

```
public void Login(string UserName, string Password) {}
```

The following code is used to call the login.

```
try
{
    Login(userName, password);
}
catch (InvalidUserNameException ex)
{...}
catch (InvalidPasswordException ex)
{...}
```

It is better to create an enumeration of possible values and then change the **Login** method to return that enumeration, as follows.

```
public enum LoginResult
{
    Success, InvalidUserName, InvalidPassword, AccountLockedOut
}
public LoginResult Login(string UserName, string Password) {}
```

The following code is used to call **Login**.

```
LoginResult result = Login(userName, password)
switch(result)
{
    case Success:
        . . .
    case InvalidUserName:
        . . .
    case InvalidPassword:
        . . .
}
```

- **Suppress the internal call to Response.End.** The **Server.Transfer**, **Response.Redirect**, **Response.End** methods all raise exceptions. Each of these methods internally call **Response.End**. The call to **Response.End**, in turn, causes a **ThreadAbortException** exception. If you use **Response.Redirect**, consider using the overloaded method and passing **false** as the second parameter to suppress the internal call to **Response.End**.

For more information, see Knowledge Base article 312629, "PRB: ThreadAbortException Occurs If You Use Response.End, Response.Redirect, or Server.Transfer," at <http://support.microsoft.com/default.aspx?scid=kb;en-us;312629>.

- **Do not catch exceptions you cannot handle.** If your code cannot handle an exception, use a **try/finally** block to ensure that you close resources, regardless of whether an exception occurs. Do not catch the exception if you cannot try recovery. Permit the exception to propagate to an appropriate handler that can deal with the exception condition.

Set Timeouts Aggressively

Page timeouts that are set too high can cause problems if parts of your application are operating slowly. For example, page timeouts that are set too high may cause the following problems:

- Browsers stop responding.
- Incoming requests start to queue.
- IIS rejects requests after the request queue limit is reached.
- ASP.NET stops responding.

The default page timeout is 90 seconds. You can change this value to accommodate your application scenario.

Consider the following scenario where an ASP.NET front-end application makes calls to a remote Web service. The remote Web service then calls a mainframe database. If, for any reason, the Web service calls to the mainframe start blocking, your front-end ASP.NET pages continue to wait until the back end calls time out, or the page timeout limit is exceeded. As a result, the current request times out, ASP.NET starts to queue incoming requests, and those incoming requests may time out, too. It is more efficient for your application to time out these requests in less than 90 seconds. Additionally, timing out the requests in less than 90 seconds improves the user experience.

In most Internet and intranet scenarios, 30 seconds is a very reasonable timeout limit. For high traffic pages such as a home page, you might want to consider lowering the timeout limit. If your application takes a long time to generate certain pages, such as report pages, increase the timeout limit for those pages.

Data Access

Almost all ASP.NET applications use some form of data access. Data access is typically a focal point for improving performance because the majority of application requests require data that comes from a database.

Use the following guidelines to improve your data access:

- **Use paging for large result sets.**
- **Use a DataReader for fast and efficient data binding.**
- **Prevent users from requesting too much data.**
- **Consider caching data.**

Use Paging for Large Result Sets

Paging large query result sets can significantly improve the performance of an application. If you have large result sets, implement a paging solution that achieves the following:

- The paging solution reduces back-end work on the database.
- The paging solution reduces the size of data that is sent to the client.
- The paging solution limits client work.

Several paging solutions are available; each solution solves the problems that are inherent to specific scenarios. The following paragraphs briefly summarize the solutions. For implementation-specific details, see the "How To: Page Records in .NET Applications" in the "How To" section of this guide.

A relatively quick and easy solution is to use the automatic paging provided by the **DataGrid** object. However, this solution works only for tables that have unique incrementing columns; it is not suitable for large tables. With the custom paging approach, you set **AllowPaging** and **AllowCustomPaging** properties to **true**, and then set the **PageSize** and **VirtualItemCount** properties. Then the **StartIndex** (the last browsed row) and **NextIndex** (**StartIndex** + **PageSize**) properties are calculated. The **StartIndex** and **NextIndex** values are used as ranges for the identity column to retrieve and display the requested page. This solution does not cache data; it pulls only the relevant records across the network.

There are several solutions available for tables that do not have unique incrementing column numbers. For tables that have a clustered index and that do not require special server-side coding, use the **subquery** solution to track the number of rows to skip from the start. From the resulting records, use the TOP keyword in conjunction with the **<pagesize>** element to retrieve the next page of rows. Only the relevant page records are retrieved over the network. Other solutions use either the **Table** data type or a global temporary table with an additional IDENTITY column to store the queried results. This column is used to limit the range of rows fetched and displayed. This requires server-side coding.

Use a DataReader for Fast and Efficient Data Binding

Use a **DataReader** object if you do not need to cache data, if you are displaying read - only data, and if you need to load data into a control as quickly as possible. The **DataReader** is the optimum choice for retrieving read-only data in a forward-only manner. Loading the data into a **DataSet** object and then binding the **DataSet** to the control moves the data twice. This method also incurs the relatively significant expense of constructing a **DataSet**.

In addition, when you use the **DataReader**, you can use the specialized type-specific methods to retrieve the data for better performance.

Prevent Users from Requesting Too Much Data

Allowing users to request and retrieve more data than they can consume puts an unnecessary strain on your application resources. This unnecessary strain causes increased CPU utilization, increased memory consumption, and decreased response times. This is especially true for clients that have a slow connection speed. From a usability standpoint, most users do not want to see thousands of rows presented as a single unit.

Limit the amount of data that users can retrieve by using one of the following techniques:

- Implement a paging mechanism. For more information, see "How To: Page Records in .NET Applications" in the "How To" section of this guide.
- Design a master/detail form. Instead of giving users all of the information for each piece of data, only display enough information to allow the users to recognize the piece of data they are interested in. Permit the user to select that piece of data and obtain more details.
- Enable users to filter the data.

Consider Caching Data

If you have application-wide data that is fairly static and expensive to retrieve, consider caching the data in the ASP.NET cache.

Security Considerations

Security and performance are often at the center of design tradeoffs, because additional security mechanisms often negatively impacts performance. However, you can reduce server load by filtering unwanted, invalid, or malicious traffic, and by constraining the requests that are allowed to reach your Web server. The earlier that you block unwanted traffic, the greater the processing overhead that you avoid. Consider the following recommendations:

- **Constrain unwanted Web server traffic.**
- **Turn off authentication for anonymous access.**
- **Validate user input on the client.**
- **Avoid per-request impersonation.**
- **Avoid caching sensitive data.**
- **Segregate secure and non-secure content.**
- **Only use SSL for pages that require it.**
- **Use absolute URLs for navigation.**
- **Consider using SSL hardware to offload SSL processing.**
- **Tune SSL timeout to avoid SSL session expiration.**

Constrain Unwanted Web Server Traffic

Constrain the traffic to your Web Server to avoid unnecessary processing. For example, block invalid requests at your firewall to limit the load on your Web server. In addition, do the following:

- Map unsupported extensions to the 404.dll file in IIS.
- Use the UrlScan filter to control the verbs and the URL requests that you allow. Verbs that you might want to control include **Get**, **Post**, and **SOAP**.
- Review your IIS logs. If the logs are full of traffic that you do not allow, investigate blocking that traffic at the firewall or filtering the traffic by using a reverse proxy.

Turn Off Authentication for Anonymous Access

Partition pages that require authenticated access from pages that support anonymous access. To avoid authentication overhead, set the authentication mode to **None** in the Web.config file in the directory that contains the anonymous pages. The following line shows how to set the authentication mode in the Web.config file.

```
<authentication mode="None" />
```

Validate User Input on the Client

Consider using client-side validation to avoid sending unwanted traffic to the server. However, do not trust client-side validation alone because it can easily be bypassed. For security reasons, you should implement the equivalent server-side checks for every client check.

Avoid Per-Request Impersonation

Per-request impersonation where you use the original caller's identity to access the database places severe scalability constraints on your application. Per-request impersonation prevents the effective use of database connection pooling. The trusted subsystem model is the preferred and scalable alternative. With this approach, you use a fixed service account to access the database and to pass the identity of the original caller at the application level if the identity of the original caller is required. For example, you might pass the identity of the original caller through stored procedure parameters.

Avoid Caching Sensitive Data

Instead of caching sensitive data, retrieve the data when you need it. When you measure application performance, if you discover that retrieving the data on a per-request basis is very costly, measure the cost to encrypt, cache, retrieve, and decrypt the data. If the cost to retrieve the data is higher than the cost to encrypt and decrypt the data, consider caching encrypted data.

Segregate Secure and Non-Secure Content

When you design the folder structure of your Web site, clearly differentiate between the publicly accessible areas and restricted areas that require authenticated access and Secure Sockets Layer (SSL). Use separate subfolders beneath the virtual root folder of your application to hold restricted pages such as forms logon pages, checkout pages, and any other pages that users transmit sensitive information to that needs to be secured by using HTTPS. By doing so, you can

use HTTPS for specific pages without incurring the SSL performance overhead across your entire site.

Avoid XCOPY Under Heavy Load

XCOPY deployment is designed to make deployment easy because you do not have to stop your application or IIS. However, for production environments you should remove a server from rotation, stop IIS, perform the XCOPY update, restart IIS, and then put the server back into rotation.

It is particularly important to follow this sequence under heavy load conditions. For example, if you copy 50 files to a virtual directory, and each file copy takes 100 milliseconds, the entire file copy takes 5 seconds. During that time, the application domain of your application may be unloaded and loaded more than once. Also, certain files may be locked by the XCOPY process (Xcopy.exe). If the XCOPY process locks certain files, the worker process and the compilers cannot access the files.

If you do want to use XCOPY deployment for updates, the .NET Framework version 1.1 includes the **waitChangeNotification** and **maxWaitChangeNotification** settings. You can use these settings to help resolve the XCOPY issues described in this section.

Note These settings are also available in a hotfix for .NET Framework version 1.0. For more information, see Knowledge Base article 810281, "Error Message: Cannot Access File AssemblyName Because It Is Being Used by Another Process," at <http://support.microsoft.com/default.aspx?scid=kb;en-us;810281>.

The value of the **waitChangeNotification** setting should be based on the amount of time that it takes to use XCOPY to copy your largest file. The **maxWaitChangeNotification** setting should be based on the total amount of time that XCOPY uses to copy all the files plus a small amount of extra time.

Few tips to improve the performance of your web application.

1. View state

View state is wonder mechanism which shows the details of entry which posted on the server. Every time its get loaded from the [server](#). This option looks like extra feature for the end users. This needs to be loaded from the server and it adds more size to the page But it will affect the performance when we have many controls in the page like user registration. So, if it is no need then it can be disabled.

EnableViewState = "false" needs to be given based on the requirement. It can be given at the control, page and config level settings.

2. Avoid Session and Application Variables

Session is storage mechanism which helps the developers to take the value across

the pages. It will be stored based on the session state chosen. By default it will be stored in the Inproc. That default settings uses the IIS. When use this Session variable in a page, which is accessed by many numbers then it will occupy more memory allocation and gives addition overhead to the IIS. It will make the slow performance.

Most of the scenarios it can be avoided. If you want to send the information across the pages then we can use Cross Post back, Query string with encrypted. If you want to store the information with in the page, then cache object is best way.

3. **Use Caching**

ASP.Net has the very significant feature of caching mechanism. It gives more performance and avoids the client/server process. There are three types of caching in ASP.Net.

If there is any static content in the full pages then it should be used the Output cache. What it does, it stores the content on IIS. When the page is request it will load immediately from the IIS for the certain period of time. Similarly Fragment paging can be used for store the part of [web page](#).

4. **Effectively use CSS and Script files**

If you have big CSS files which used for the entire site different pages, then based on the requirement it can be spitted and stored with different names. It will minimize the loading time of the pages.

5. **Images sizes**

Over use of images in the [web site](#)

affect the web page performance. It takes time to load the images especially on dial up connections. Instead of using the background images, it can be achieved on the CSS colors or use light weight images to repeat in the entire pages.

6. **CSS based layout**

The entire [web page design](#) controlled by the CSS using the div tags instead of table layout. It increases the page loading performance dramatically. It will help to enforce same standard guideline throughout the website. It will reduce the future changes easily. When we use the nested table layout it takes more time for rendering.

7. **Avoid Round trips**

We can avoid unnecessary database hits to load the unchanged content in the database. We should use IsPostBack method to avoid round trips to database.

8. **Validate using JavaScript**

Manual validation can be done at the client browser instead of doing at the server side. The [JavaScript](#) help us to do the validation at the client side. This will reduce the

additional overhead to the server.

The plug-in software helps to disable the coding in the client browser. So, the sensitive application should to the server side validation before go into the process.

9. Clear the Garbage Collection

Normally .Net application use the Garbage collection to clean the unused the resources from the memory. But it takes own time to clear the unused objects from the memory.

There are lots of ways to clean the unused the resource. But not all the methods recommended. But we can use dispose method in the finally block to clean the resources. More over we have to close the connection. It will immediately free the resources and gives spaces in the memory.

10. Avoid bulk data store on client side

Try to avoid more data on the client side. It will affect the web page loading. When we store more data on the hidden control then it will encrypt and store on the client side. It can be tamper by hackers as well.

11. Implement Dynamic Paging

When we load bulk number of records in server data controls like GridView, DataList, and ListView it will take time to load. So we can show only the current page data through the dynamic paging.

12. Use Stored Procedure

Try to use stored procedure maximum. It will increase the performance of the web pages. Because it is stored as compiled object in the database and it uses the query execution plans. If you pass the query then it will make network query. In the stored procedure single line will be passed to the backend.

13. Use XML and XSLT

XML and XSLT will speed up the page performance. If the process is not more complex then it can be implemented in XSLT.

14. Use Dataset

The DataSet is not light weight when compare with DataReader. But it has the advantages of disconnected oriented architecture. The DataSet will consume lot of memory. Even though it can have more than one day. If you want to perform many operations while loading the page itself, then you can better go with DataSet. Once data is loaded into DataSet then it can be used later also.

15. Use String Builder in place of String

When we append the strings like mail formatting in server side, then we can use String Builder. If you use string for concatenation, what it does every time it create

the new storage location for place that string. It occupies more spaces in memory. But if we use String Builder class in C# it consumes more memory space than String.

16. Use Server.Transfer

If you want to transfer the page with in the current server then we can use the Server.Transfer method. It avoids roundtrips between the browser and server. But it won't update the browser history.

17. Use Threads

Thread is important mechanism in the [programming language](#) to utilize the system resources effectively. When we want to do background process then it can be called as background process.

Consider an example, when click on send, it should sends the mail to 5 lakhs members that time no need to wait for all the process completion. Just call the mail sending process as background thread then go ahead for the further process. Because this mail sending not depend on the any other process.